NumLin: Linear Types for Numerical Libraries

Dhruv C. Makwana$^{1[orcidID]}$ and Neelakantan R. Krishnaswami$^{2[orcidID]}$

[1] dcm41@cam.ac.uk dhruvmakwana.com
[2] Department of Computer Science, University of Cambridge
nk480@cl.cam.ac.uk

Abstract. Briefly summarize the contents of the paper in 150–250 words.

Keywords: numerical, linear, algebra, types, permissions, OCaml

1 Introduction

NumLin is a functional programming language designed to express the APIs of low-level linear algebra libraries (such as BLAS/LAPACK) safely and explicitly. It does so by combining linear types, fractional permissions, runtime errors and recursion into a small, easily understandable, yet expressive set of core constructs. In addition to this, NumLin's implementation supports several syntactic conveniences as well as a usable integration with real-world OCaml code.

1.1 Contributions

- We have designed the NumLin programming language
- We illustrate that the design is sensible with many matrix-y examples
- We give a soundness proof for NumLin, using a step-indexed logical relation
- Incredibly simple type inference algorihtm for polymorphic fractional permissions
  - Compare to Bierhof et al's *Fraction Polymorphic Permission Inference*, which uses a fancy dataflow analysis
  - We use exactly the same unification algorithm type polymorphism does
- We have an implementation - compatible with and usable from existing code!

2 NumLin Overview and Examples

2.1 Overview

Linearity is at the heart of NumLin. Linearity allows us to express a pure-functional API for numerical library routines that mutate arrays and matrices. Linearity also restricts aliasing of (values which represent) pointers.

Intuitionism: ! and `Many` However, linearity by itself is not sufficient to produce an expressive enough programming language. For values such as booleans, integers, floating-point numbers as well as pure functions, we need to be able to use them *intuitionistically*, that is, more than once or not at all. For this reason, we have the ! constructor at the type level and its corresponding `Many` constructor and `let Many <id> = .. in ..` eliminator at the term level. Because we want to restrict how a programmer can alias pointers and prevent a programmer from ignoring them (a memory leak), NumLin enforces simple syntactic restrictions on which values can be wrapped up in a `Many` constructor (details in Section 3).

Fractional Permissions There are also valid cases in which we would want to alias pointers to a matrix. The most common is exemplified by the BLAS routine `gemm`, which (rather tersely) stands for *GEneric Matrix Multiplication*. A *simplified* definition of `gemm(`α`, A, B, `β`, C)` is $C := \alpha AB + \beta C$. In this case, `A` and `B` may alias each other but neither may alias `C`, because it is being written to. Related to *mutating* arrays and matrices is *freeing* them. Here, we would also wish to restrict aliasing so that we do not free one alias and then attempt to use another. Although linearity on its own suffices to prevent use-after-free errors when values are *not* aliased (a freed value is *out of scope* for the rest of the expression), we still need another simple, yet powerful concept to provide us with the extra expressivity of aliasing *without* losing any of the benefits of linearity.

Fractional permissions provide exactly this. Concretely, types of (pointers to) arrays and matrices are *parameterised* by a *fraction*. A fraction is either 1 (2^0) or exactly *half* of another fraction (2^{-k}, for natural k). The former represents complete ownership of that value: the programmer may mutate or free that value as they choose; the latter represents read-only access or a *borrow*: the programmer may read from the value but not write to or free it. Creating an array/matrix gives you ownership of it, so too does having one (with a fractional permission of 2^0) passed in as an argument.

In NumLin, we can produce two aliases of a single array/matrix, by *sharing* it. If the original alias had a fractional permission of 2^{-k} then the two new aliases of it will have a fractional permission of $2^{-(k+1)}$ each. Thanks to linearity, the original array/matrix with a fractional permission of 2^{-k} will be out of scope after the sharing. When an array/matrix is shared as such, we can prevent the programmer from freeing or mutating it by making the types of `free` and `set` (for mutation) require a *whole* (2^0) permission.

If we have two aliases *to the same matrix* with *identical* fractional permissions ($2^{-(k+1)}$), we can recombine or *unshare* them back into a single one, with a larger 2^{-k} permission. As before, thanks to linearity, the original two aliases will be out of scope after unsharing.

Runtime Errors Aside from out-of-bounds indexing, matrix unsharing is one of only *two* operations that can fail at runtime (the other being dimension checks, such as for `gemm`). The check being performed is a simple sanity check that the two aliasing pointers passed to `unshare` point to the same array/matrix. Section 5 contains an overview of how we could remove the need for this by tracking pointer identities statically by augmenting the type system further.

Recursion The final feature of NumLin which makes it sufficiently expressive is recursion (and of course, conditional branches to ensure termination). Conditional branches are implemented by ensuring that both branches use the same set of linear values. A function can be recursive if it captures no linear values from its environment. Like with `Many`, this is enforced via simple syntactic restrictions on the definition of recursive functions.

2.2 Examples

The more examples, the better. In fact, it's actually impossible to have too many examples – it's okay (indeed, desirable) to spend 5-6 pages on examples.

- Simple: factorial, shows recursion, !-annotations etc.
- Less simple: summing over an array, indexing and safety
- Medium: one-dimensioal convolution, permissions
- Harder: linear regression, pattern-matching and apparent non-linearity
- Harder: L1 norm-minimisation, some frees, re-using memory
- Big finish: Kalman filter

Introductory: Factorial Although a factorial function (Figure 1) may seem like an aggresively pedestrian first example, in a linearly typed language such as NumLin it represents the culmination of many features.

To simplifiy the design and implementation of NumLin's type system, recursive functions must have full type annotations (non-recursive functions need only their argument types annotated). Its body is a closed expression (with respect to the function's arguments), so it type-checks (since it does not capture any linear values from its environment).

The only argument is `!x : !int`. The !-annotation on `x` is a syntactic convenience for declaring the value to used intuitionistically, its full and precise meaning is described in Section 4.

The condition for an `if` may or may not use linear values (here, with `x < 0 || x = 0`, it does not). Any linear values used by the conditional would not be in scope in either branch of the `if`-expression. Both branches use `x` differently: one ignores it completely and the other uses it twice.

All numeric and boolean literals are implicitly wrapped in a `Many` and all primitives involving them return a `!int`, `!bool` or `!elt` (types of elements of arrays/matrices, typically 64-bit floating-point numbers). The short-circuting `||` behaves in exactly the same way as a boolean-valued `if`-expression.

```
let rec f ( !x : !int ) : !int =
    if x < 0 || x = 0 then
        1
    else
        x * f (x - 1) in f
;;
```

Fig. 1. Factorial function in NumLin.

```
let rec f (!i : !int) (!n : !int) (!x0 : !elt)
          ('x) (row : 'x arr) : 'x arr * !elt =
    if i = n then
        (row, x0)
    else
        let (row, !x1) = row[i] in
        f (i + 1) n (x0 +. x1) 'x row in
    f
;;
```

Fig. 2. Summing over an array in NumLin.

Introductory: Summing over an Array Now we can add fractional permissions to the mix: Figure 2 shows a simple, tail-recursive implementation of summing all the elements in an array. There are many new features; first among them is `!x0 : !elt`, the type of array/matrix elements (64-bit floating point).

Second is `('x) (row : 'x arr)` which is an array with a universally-quantified fractional permission. In particular, this means the body of the function cannot mutate or free the input array, only read from it. If the programmer did try to mutate or free `row`, then they would get a helpful error message (Figure 3).

Alongside taking a `row : 'x arr`, the function also returns an array with exactly the same fractional permission as the `row` (which can only be `row`). This is necessary because of linearity: for the caller, the original array passed in as an argument would be out of scope for the rest of the expression, so it needs to be returned and then rebound to be used for the rest of the function.

An example of this consuming and rebinding is in `let (row, !x1) = row[i]`. Indexing is implemented as a primitive `get: 'x. 'x arr --o !int --o 'x arr * !elt`. Although fractional permissions can be passed around explicitly (as done in the recursive call), they can also be *automatically inferred at call sites*, `row[i] == get _ row i` takes advantage of this convenience.

3 Formal System

3.1 The Core Type Theory and Dynamic Semantics

Describe the typing rules and operational semantics here

```
let row = row[i] := x1 in (* or *) let () = free row in
(* Could not show equality:                                        *)
(*     z arr                                                       *)
(* with                                                            *)
(*     'x arr                                                      *)
(*                                                                 *)
(* Var 'x is universally quantified                                *)
(* Are you trying to write to/free/unshare an array you don't own? *)
(* In test/examples/sum_array.lt, at line: 7 and column: 19        *)
```

Fig. 3. Attempting to write to or free a read only array in NumLin.

3.2 The logical relation

Describe the step-indexed logical relation and its main properties

3.3 Soundness Theorem

State the fundamental lemma, and sketch the proof a little

4 Implementation

4.1 Implementation Strategy

Talk about how you implemented NumLin and the general architecture. Talk about how simple everything is, and also about how implementing inference for fractions is.

4.2 Performance Metrics

Here, evaluate the performance of the examples from the second section. Compare with your C implementations, and perhaps as well as the straightforward math transcribed into (Matlab/R/Numpy?).

5 Discussion and Related Work

The main point we want to make is that using linear types for BLAS is an "obvious" idea, but is surprisingly under-explored.

- Rust
- ATS
- Single-assignment C
- Linear Haskell
- Bernardy and Sveningsson
- L3
- Boyland fractional permissions

References